```cpp
//--------------------------------------------------------------------------------------------------------------------
// 1. INTRODUCTION
//--------------------------------------------------------------------------------------------------------------------
        #include <iostream>
        using namespace std;

//--------------------------------------------
// STRINGS
//--------------------------------------------
        #include<string>
        using std::string;

// * strings have variable-length: string first_name; NOT: string first_name[100];
// * operations:
        string s0, s1;
        std::cin >> s0 >> s1; // defined by user
        string s2 = s1;       // s2 is initialized as copy of s1
        string s3 = "hi";     // s3 is a copy of the string literal (copy init)
        string s4("hi");      // s4 is directly initialized (direct init)
        string s5(5, 'c');    // s5 is ccccc (note: it's 'c', not "c") (direct init)

        string s6 = s3 + " " + s4;        // s6 is "hi hi"
        string s7 = s3 + " ";             // ok, adding a literal
        string s8 = "hello" + " ";        // ERROR: no string operand
        string s9 = s3 + " " + "hi";      // ok
        string s10 = "hello" + " " + s3;  // ERROR: can't add string literals


        s0.empty(); // returns True if the string is empty
        s0.size();  // returns the number of character of the string
        s0[n];      // returns a reference to the char at position n
        s0 += s1;   // adds the string s1 at the end of s0
        s0 == s1;
        s0 != s1;
        s++ / s-    // ERRORs

// * we can use range-based for loop:
        string str("some string");
        for( auto c:str){ std::cout << c << endl; }

// Example (getline(..,..))
        string line;
        while(getline(cin, line)){
            if(!line.empty() && line.size()<100)
                ..line..;
        }

//--------------------------------------------
// std::cin
//--------------------------------------------
        int value;
        while (cin>>value){
            ..
        }
// std::cin returns True/False. If we take as input what we expect (value is an int) -> True.
// If, for example, it gets a string, then the while ends.

        using std::cin;
        using std::cout;
        using std::endl;

//--------------------------------------------
// INCREMENTS
//--------------------------------------------
        int i=0;
        a=i++;    // 1.
        b=++i;    // 2.

// a=0; b=2;, i=2;
// 1. initializes a to the value of i and then it increments a.
// 2. first increments the value of i, then it initializes b with the just-updated value.

//--------------------------------------------
// TYPE CONVERSIONS
//--------------------------------------------
        bool b=42;     // b is true (when we assign something != 0 -> we get True
        int i=b;       // i = 1
        i=3.14;        // i = 3
        double pi=i;   // pi = 3.0

//--------------------------------------------
// EXPRESSIONS
//--------------------------------------------
//  * == : equal
//  * != : not equal
//  * && : and
//  * || : or
//  * !  : not
//  * </>/<=/>=
//  * ?: : ternary condition
```

```
//------------------------------------------
// ITERATIONS
//------------------------------------------
        while(..i..){
            ..i..
        }
// while(condition){ variation; }

        for(unsigned i=0; i..; i++){}
        for(size_t i=0; i..; i++){}
// for(initialize; condition; variation);

//------------------------------------------
// ARRAYS
//------------------------------------------
        const unsigned sz = 3;
        int vec1[sz];                   // randomly initialized
        int vec2[sz] = {};              // initialized with 0s
        int vec3[sz] = {0,1,2};
        int vec4[] = {0,1,2};
        int vec5[5] = {0,1,2};          // same as vec5[] = {0,1,2,0,0}
        string vec6[3] = {"hi", "bye"};   // same as vec6[] = {"hi", "bye", ""}

// * dimension must be known at compile time: we cannot add elements.
// * declaration: base_type array_name[size];
// * no assignment: int vec7[] = vec1; ERROR -> we need to do it elementwise
// * no copy: int vec7[]; vec7 = vec1; ERROR -> we need to do it elementwise

// MATRICES - arrays of arrays
        int rows, columns;
        int matrix[rows][columns];

        int mat1[3][4];                 // randomly initialized
        int mat2[3][4] = {};            // initialized with 0s
        int mat3[2][4] = {{0,1,2,3}, {4,5,6,7}};
        int mat4[2][4] = {0,1,2,3,4,5,6,7};

// mat3 and mat4 are equivalent

//------------------------------------------
// STRUCTS
//------------------------------------------
        struct Student{
            string last_name;
            string name;
            unsigned id;
            unsigned grades[2];
        };

        Student s;                  // declaration
        s.name = "Name";            // assignment
        s.last_name = "Last Name"
        s.grades[0] = 30;
        s.grades[1] = 29;

// var_name.field_name

// * we can globally assign:
        Student s1, s2;
        s1. .. = ..;
        s2 = s1;
// * we CANNOT compare (==): we must compare field by field
// * we can create arrays of structs
        Student classroom[10];
        classroom[0].grades[0];  // 1st grade of the 1st student

//------------------------------------------
// VECTORS
//------------------------------------------
// Vectors are variable-sized arrays. We don't need to specify their length.
// A vector is a class template (blueprint type-independent).

        #include<vector>
        using std::vector;

        int main(){
            vector<double> temps;
            double temp;
            while(cin>>temp){             // we go on untill we get something != double
                temps.push_bacl(temp);    // we attatch the new double (add element to the vector)
            }
            return 0;
        }

// * we can access each element by the index (like in the array case): "temps[0]" = 1st elem of temps
// * simpler way of looping (range-for loop):
        vector<data_type> vec;
        ..vec..;
        for(size_t i=0; i<vec.size(); ++i){ .. }
        for(data_type x : vec){ .. }
```

```cpp
// * ways to initialize a vector:
        vector<int> v1;           // v1 is empty
        vector<int> v2(v1);       // v2 has a copy of each element in v1
        vector<int> v2=v1;        // v2 has a copy of each element in v1
        vector<int> v3(n,val);    // v3 has n elements, all val
        vector<int> v4(n);        // v4 has n elements, all default initialized
        vector<int> v5{a,b,c};    // v5 has 3 elements, the values of a,b,c
        vector<int> v5 = {a,b,c}  // v5 has 3 elements, the values of a,b,c
// * operations:
        vector<int> vec, vec2;
        vec.empty();          // returns True if the vector is empty
        vec.size();           // returns the number of elements of the vector
        vec.push_back(elem);  // adds an element with value elem in the vector
        vec[n];               // returns a reference to the element at position n
        vec == vec2;
        vec != vec2;
// * ERROR -ATTENTION-
        vector<int> vec;
        std::cout << vec[0]; // ERROR: vec is the empty vector!
// Example: word list
        vector<string> words;
        for(string s; cin>>s && s!="quit"; )  // go on until we don't get "quit"
            words.push_back(s);

//--------------------------------------------
// Random REMEMBER
//--------------------------------------------
// 1. Declare a function!
//     To refer to something we need (only) its DECLARATION.
//     Where do we place delcarations? In HEADER FILES.
//     (The primary purpose of head file is to propagate declarations to code files)
        #include<iostream>
        using namespace std;
        int function();
        int main(){
                ...
        }
        int function(){
                ...
        }

// 2. We CANNOT define something twice, we can declare something twice.
        double function(double d){..}   // ok
        double function(double d){..}   // ERROR

        double function(double);        // ok (declaration)
        double function(double d){..}   // ok (definition)

// 3. Vector are classes, arrays are pointers.
        int a[10], b[10];
        for(unsigned i=0; i<10; i++){ a[i] = b[i]; }  // there's no a=b;
        vector<int> aa(10), bb(10);
        aa = bb;                                       // there is aa=bb;

// 4. Shortcut
        enum ZoneType{EST,CST,MST,PST,EDT,CDT,MDT,PDT};
//     In this way we enumerate the values that we can use to assign.
//     Here we're saying EST=0, CST=1, ..
        ZoneType var;     // equivalent to:   int var;
        var = MST;        //                  var = 2;
```

```
//--------------------------------------------------------------------------------------------------------------------
// 2. CLASSES
//--------------------------------------------------------------------------------------------------------------------

//-------------------------------------------
// HEADER FILES
//-------------------------------------------
// The purpose is to propagate delcarations to code file (.cpp).
// To use an external code file we need to include its header.

// Header file: "MyFriendLibrary.h"
        #ifndef MY_FRIEND_LIBRARY_H  // "if MY_FRIEND_LIBRARY_H is not defined yet, then.."
        #define MY_FRIEND_LIBRARY_H
        void function();
        #endif // MY_FRIEND_LIBRARY_H

// Source file: "MyFriendLibrary.cpp"
        #include<iostream>
        #include "MyFriendLibrary.h"
        void function(){
            // definition
        }

// At compile time: "MyFriendLibrary.cpp"
        #include<iostream>
        #define MY_FRIEND_LIBRARY_H
        void function();
        void function(){
            // definition
        }

// We want to use "function" defined in the main and "function" of this library? Namespaces!
        #include<iostream>
        #include "MyFriendLibrary.h"
        namespace nuovo{ void function(); }
        int main(){
            nuovo::function();      // this calls the function defined below
            function();             // this calls the function from "MyFriendLibrary.h"
        }
        function(){..}

//-------------------------------------------
// CLASSES - pt. 1
//-------------------------------------------
// Classes refers to a blueprint. We define data members and methods the objects support.
// Objects are instances of classes. Objects have states which can be changed by some methods.
        class X{
            public:
                int m;
                int mf(int v){int old=m; m=v; return old;}
        };

        X var;
        var.m = 7;              // var.m == 7;
        int val = var.mf(9);    // var.m == 9;

// * if a method is CONST then it doesn't modify the state of the object
// * members and methods can be public, private or protected
// * to get data which are private we need to use getters (const methods) and setters
// * .h is for the class declaration, .cpp is for the class definition
// * in the .cpp we don't have always to specify where the method is from, i.e.
//      "Example.h"
        class Example{
            public:
                Example(){ value = -1; }       // constructor (it initializes value to -1)
                int function1() const;         // for sure it doesn't modify the state
                int function2();
                ~Example(){ std::cout<<"Ok";}  // destructor
            private:
                int value;
        };
//      "Example.cpp"
        int Example::function2(){
            int x = function1();   // we don't need to write Example::function1();
            ..
        }

//-------------------------------------------
// CLASSES - pt. 2
//-------------------------------------------
// POINTERS AND CLASSES
// * members are accessing using .(dot) for objects and -> (arrow) for pointers
        class Example{
            public:
                int value;
        };

        Example x;
        Example* p = &x;
        x.value    = 7;
        (*p).value = 7;
        p->value   = 7;
```

```
// CLASSES AND "const"
// * we can run only constant method on constant objects
//   (we cannot run a method which modify the state of a constant object, and so
//    we CANNOT run a non-const method on a const object)
        class Example{
            private:
                int value;
            public:
                Example(int val): value(val){}
                void function_1() const;
                void function_2();
        };

        const Example x(5);
        Example y(4);
        x.function_1();       // ok
        x.function_2();       // ERROR: the method is NOT const
        y.function_1();       // ok
        y.function_2();       // ok

//------------------------------------------
// CONSTRUCTORS (and DESTRUCTOR)
//------------------------------------------
// To define the initial state of the objects we must define a constructor. We may have multiple
// constructors (overload method). Note that constructors have no return value.
// The destructor operates inversely to the constructors and is automatically invoked every time
// an object goes out of scope. The prefix of the destructor is ~.

// CONSTRUCTORS
// If we don't specify any constructor, it will be implicitly defined by the compiler.
// The default constructor takes no arguments. If there is an in-class initialization,
// the default constructor uses it to initialize the members, otherwise it default-initializes them.
// If we provide a constructor, the provider constructor will cover the default one.
// However it's always convenient to provide a default constructor, even if others are being defined.

// ---Example 1---
//    Example.h
        class Example{
            private:
                int value_1 = 0;    // the "=0" is an in-class initialization
                int value_2 = 1;    // the "=1" is an in-class initialization
            public:
                Example() = default;
                Example(int v1, int v2);
                /* getters, setters */
        };

//    Example.cpp
        Example::Example(int v1, int v2): value_1(v1), value_2(v2) {}

//    main.cpp
        Example e1;       // initialized with: value_1=0, value_2=1 -> we need setters to be able to change it
        Example e2(2,1);  // initialized with: value_1=2, value_2=1

// ---Example 2---
        class Sales{
            private:
                std::string Book;
                unsigned unit_sold = 0;
                double revenue = 0.0;
            public:
                Sales() = default;
                Sales(const std::string& s): Book(s) {}
                Sales(const std::string& s, unsigned n, double p): Book(s), unit_sold(n), revenue(p*n){}
        };
//    The 1st constructor (default) initializes the Book as empty string, unit_sold=0, revenue=0.0.
//    The 2nd constructor initializes the Book as s, unit_sold=0, revenue=0.0.
//    The 3rd constructor initializes the Book as s, unit_sold=n, revenue=p*n (price*#books).

// CONSTRUCTOR AND "this"
// "this" is a const pointer initialized with the address of the object on which the function was invoked
        class Example{
            private:
                int value_1;
                int value_2;
            public:
                Example(int value_1, int value_2);
                void setValues(int value_1, int_value_2);
        };

        Example::Example(int value_1, int value_2){
            this -> value_1 = value_1;   // we CANNOT simply do "value_1 = value_1"
            this -> value_2 = value_2;
        }

        void Example::setValues(int value_1, int_value_2){
                this -> value_1 = value_1;
            this -> value_2 = value_2;
        }
```

```
// CONSTRUCTOR INITIALIZER LIST vs BODY INITIALIZER
// There is a difference when we define and initialize vs. when we default initialize and assign.
        string s1 = "Hello World!";     // defined and initialized
        string s2;                      // default initialized to an empty string
        s2 = "Hello World!";            // assigned
// If we have a class called "Example", which has only two private members "val_1" and "val_2",
// we can provide a list-init constructor and a body-init constructor:
        Example::Example(int v1, int v2): val_1(v1), val_2(v2){}
        Example::Example(int v1, int v2){ val_1 = v1; val_2 = v2; }
// The first (list-init) constructor is comparable to what we did with string s1.
// The second (body-init) constructor is comparable to hat we did with string s2.
// However the body-init constructor is not always an option:
        class Exception{
            private:
                int i;
                const int ci;
                int& ri;
            public:
                Exception(int val);
        };

        Exception::Exception(int val){
            i = val;
            ci = val;   // ERROR: cannot assign to a constant
            ri = i;     // ERROR: cannot assign values to a reference
        }

// Here the only available option is
        Exception::Exception(int val): i(val), ci(val), ri(i) {}


// DELEGATING CONSTRUCTORS
// We can re-use a constructor to define other constructors.
        class Sales{
            private:
                std::string Book;
                unsigned unit_sold;
                double revenue;
            public:
                Sales(const std::string& s, unsigned n, double p): Book(s), unit_sold(n), revenue(p*n){}
                Sales(const std::string& s): Sales(s, 0, 0.0){}
                Sales(): Sales("0", 0, 0.0){}
        };

//--------------------------------------------
// HELPER FUNCTIONS and OPERATORS OVERLOAD
//--------------------------------------------
// To keep the class interface simple and minimal, we need extra "helper functions" outside the class.
//   -> declare helper functions in the class header (.h) file
//   -> define helper functions in the class source (.cpp) file
// For example, we may need operators-ad-hock: operator overloading!

//      Example.h
        class Example{ .. };
        bool operator==(const Example&, const Example&);
        bool operator!=(const Example&, const Example&);
        Example operator+(const Example&, const Example&);

//      Example.cpp
        Example:: ..
        bool operator==(const Example& a, const Example& b){ .. }
        bool operator!=(const Example& a, const Example& b){ .. }
        Example operator+(const Example& a, const Example& b){ .. }

// Note: to overload an operator we must have at least one user-defined type as operand
        int operator+(int,int);  // ERROR

// OPERATORS OVERLOAD - IN-CLASS
// We may as well add ad-hock operators as methods of the class.

//      Example.h
        class Example{
            private:
                vector<int> v;
            public:
                /* getters, setters */
                Example operator+(const Example& w) const;
                int operator[](unsigned n) const;
        };

//      Example.cpp
        Example Example::operator+(const Example& w) const{
            Example result;
            .. ;
            return result;
        }

        int Example::operator[](unsigned n) const{
            int result;
            .. ;
            return result;
        }
```

```
// Thanks to this we can write
        Example e3 = e1 + e2;
        Example e3 = e1.operator+(e2);
        e3[0] = 4;

// IN-CLASS vs. OUT-CLASS
// Why should we prefer the in-class or the out-class version of the operator overloading?
// Because the usage and the implementation is different!
// IN-CLASS
        class Version1{
            private:
                int value_1;
                int value_2;
            public:
                void set_value_1(v1);
                void set_value_2(v2);
                int get_value_1();
                int get_value_2();
                Version1 operator+(const Version1& rhs) const;
        };

        Version1 Version1::operator+(const Version1& rhs) const{
            Version1 result;
            result.value_1 = value_1 + rhs.value_1;
            result.value_2 = value_2 + rhs.value_2;
            return result;
        }

// we can write:
        Version1 ver3 = ver1 + ver2;
        Version1 ver3 = ver1.operator+(ver2);

// OUT-CLASS
        class Version2{
            private:
                int value_1;
                int value_2;
            public:
                void set_value_1(v1);
                void set_value_2(v2);
                int get_value_1();
                int get_value_2();
        };

        Version2 operator+(const Version2& lhs, const Version2& rhs);

        Version2 operator+(const Version2& lhs, const Version2& rhs){
            Version2 result;
            result.set_value_1( lhs.get_value_1() + rhs.value_1 );
            result.set_value_2( lhs.get_value_2() + rhs.value_2 );
            return result;
        }

// DEFINING A FUNCTION TO RETURN "this" OBJECT
        class Example{
            private:
                int value_1;
                int value_2;
            public:
                Example& operator+=(const Example& rhs);
        };

        Example& Example::operator+(const Example& rhs){
            value_1 += rhs.value_1;
            value_2 += rhs.value_2;
            return *this;
        }

//--------------------------------------------
// FRIENDS
//--------------------------------------------
// A class can allow another class or function to access its nonpublic memebers
// by making that class or function a friend.

        class Example{
            // in-class declaration of the friend (declaration of the friendship)
            friend Example operator+(const Example& lhs, const Example& rhs);
            private:
                int value_1;
                int value_2;
            public:
                void set_value_1(v1);
                void set_value_2(v2);
                int get_value_1();
                int get_value_2();
        };
        // out-class declaration (declaration of the function)
        Example operator+(const Example& lhs, const Example& rhs);
```

```
// Because of the friendship declaration, we can write the code as:
        Example operator+(const Example& lhs, const Example& rhs){
            Example result;
            result.value_1 = lhs.value_1 + rhs.value_1;
            result.value_2 = lhs.value_2 + rhs.value_2;
            return result;
        }

// Without the friendship we would have to rely on getters and setters:
        Example operator+(const Example& lhs, const Example& rhs){
            Example result;
            result.set_value_1( lhs.get_value_1() + rhs.value_1 );
            result.set_value_2( lhs.get_value_2() + rhs.value_2 );
            return result;
        }

// Notice that, because of this reason A HELPER IS NOT A FRIEND!
// A helper cannot access the private members directly, a friend can.

//-------------------------------------------
// STATIC MEMBERS
//-------------------------------------------
// Classes may have members that are associated with the class itself, not the individual objects.
// To say that a member is associated with the class, we add the keyword "static" to its declaration.

        class Account{
            private:
                std::string owner;
                double amount;
                static double interest_rate;                        // 1.
                static double init_rate();                          // 2.
            public:
                void calculate() { amount += amount*interest_rate; }    // 3.
                static double rate() { return interest_rate; }      // 4.
                static void rate(double);                           // 5.
        };

// The interest_rate (1.) is associated to the class, not with individual objects -> static.
// We say that functions are static if they rely ONLY on static members (2., 4., 5.).
// The function "calculate()" (3.) relies also on the amount -> not static.
// We can have getters and setters for a static member (4., 5., overloaded method).

// Properties:
// * static member functions are not bounded to any object
// * static member functions do NOT have "this" pointer (since it's a pointer to the underlying
//   object on which we're running the function, but static functions are NOT associated with objects)
// * static members/functions do NOT have "const" (because "const" is associated with objects)
// * we can access static member directly through the scope operator
        double r = Account::rate();
// * we can access static member also through objects
        double r;
        Account a;
        Account* p = &a;
        r = a.rate()        // through the object
        r = p->rate();      // through a pointer to the object
// * when we define a static member outside the class we do not repeat the static keyword
//   (the keyword appears only with the declaration inside the class body)
//   [considering the above declaration of the class, in Account.cpp we have:]
        void Account::rate(double new_rate){
            interest_rate = new_rate
                }
// * since static data members are not part of individual objects, they're not defined when
//   we create an object of the class -> they're not initialized by the class constructor
// * like global objects (!), static data members are defined outside any function

//-------------------------------------------
// Random REMEMBER
//-------------------------------------------
// 1. Wherever it is, a constructor need curly braces ({})!

// 2. Type aliases
//    A type alias is a name that is a synonim for another type.
//    We can define a type alias in two ways:
        typedef double new_name;   // typedef <name_of_the_type> <name_of_the_alias>;
        using new_name = double;   // using <name_of_the_alias> = <name_of_the_type>;
//    In both cases, from these lines on "new_name" is a synonym of "double".
        typedef double new_name, *p;
//    In this way "new_name" is a synonym of "double" and "p" is a synonym of "double*".
//    If we use type aliases in classes, we have to define them BEFORE we use them.

// 3. Scope
//    Classes are an exeption: class members can be used within the class before they are declared
        class Example{
            public:
                int largest_elem(){ .. vec .. }
            private:
                vector<int> vec;
        };
```

```
//--------------------------------------------------------------------------------------------------------------------
// 3. POINTERS, REFERENCES, FUNCTION PARAMETERS, ITERATORS
//--------------------------------------------------------------------------------------------------------------------

//-------------------------------------------
// POINTERS - pt. 1
//-------------------------------------------
// Declaring a variable means reserving a memory area.
// The name of the variable indicates the contents of the memory location.
// The operator "&" allows obtaining the memory address of the location associated
// with the variable to which the operand is applied.
        int var;
// var : indicates the content of the memory location
// &var: indicates the memory address

// Pointers store memory addresses.
        int *p;  // p is a pointer to an integer
        *p = 5;  // dereferencing: we access the memory location whose address is store on p and we're putting 5
         p = 5;   // ERROR

// * ATTENTION at declaration and dereferenciation
//   Case 1:
        int  x;
        int *p;
        x = 5;
        p = &x;  // x=5, *p=5, p=0x6ffe0c, &x=0x6ffe0c
//   Case 2:
        int  x = 3;
        int  y = 5;
        int *p = &x;
        *p = 7;
         p = &y;
        *p += 3;  // x=7, y=8, *p=8

//-------------------------------------------
// PASS-BY-VALUE/REFERENCE
//-------------------------------------------
// 1. PASS-BY-VALUE
//    At the time of the call the value of the actual parameter is copied.
//    The formal parameter and the actual parameter refer to two different memory locations.
//    The actual parameters are NOT changed.
        int main(){
            int a=2;
            int b=3;
            swap(a, b);
        }
        void swap(int p, int q){
            int temp = p;
            p = q;
            q = temp;
        }
//    Mechanism: we copy (a,b) in (p,q), we swap p and q while a and b remain unchanged.

// 2. PASS-BY-REFERENCE
//    At the time of the call the address of an actual parameter is associated with the formal parameter.
//    The actual parameter and the formal parameter share the same memory location.
//    Each change of the formal parameter is reflected to the corresponding actual parameter.
//    To implement pass by reference we rely on pointers (note: arrays are always passed by reference).
        int main(){
            int a=2;
            int b=3;
            swap(&a, &b);
        }
        void swap(int *p, int *q){
            int temp = *p;
            *p = *q;
            *q = temp;
        }

//-------------------------------------------
// REFERENCES
//-------------------------------------------
// If we introduce a reference for an object, we're introducing another name for the same object.
// A reference MUST be initialized when defined (unlike pointers) and CANNOT change.
        int x = 9;
        int y = 7;
        int &r = x;  // in this way, r and x are THE SAME THING
        r = 10;      // equivalent to: x=10;
        r = &y;      // ERROR: whenever we define a reference we don't change

// We can initialize a reference with a binding
        int x = 9;
        int &r1 = x;
        int &r2 = r1;  // now both r1 and r2 are the same thing as x

// * symbols and context
        int i = 42;
        int &r = i;  // r is a reference
        int *p;      // p is a pointer
        p = &i;      // p points at the address of i
        *p = i;      // dereferenciation of p
```

```
// * references and "const": const reference to a non-const variable
//   (this is good: with pointers we pass the actual parameter, an error in the function
//    can modify the parameter everywhere, instead, if we use const reference to variables
//    we have the guarancee that the variables are not modified (while they're still not copied))
        int x = 2;
        const int &r = x;  // ok: const reference to a non-const integer
        r = 4;             // ERROR: r is constant
        x = 4;             // ok: x is non-const, r becomes 4

// * references and "const": non-const reference to a const variable
        const int x = 2;
        const int &r = x;  // ok
        int &r2 = x;       // ERROR: x is constant and it NEEDS const references

// * [easier than pointer] "swap" example using references
        int main(){
            int a=2;
            int b=3;
            swap(a, b);
        }
        void swap(int &p, int &q){
            int temp = p;
            p = q;
            q = temp;
        }

// REFERENCES CANNOT BE STORED IN A VECTOR
//    References are not objects, hence we may not have a pointer to a reference.
//    We cannot have a vector of references (even simply) because we cannot store an unchangeable
//    variable into a container that allows, for example, assignemnt. With vectors we can do "v1=v2",
//    however, once a reference has been binded with a variable it cannot change!
//    Because of this incongruency -> NO! vector<type&> NO!

//-------------------------------------------
// POINTERS - pt. 2
//-------------------------------------------
// RAW POINTERS AND MEMORY LEAK
// Is there "new"? There must be a "delete"!
// We allocate a piece of memory which will remain allocated until a delete or the end of the program.
        int *p = new int;     // allocates one uninitialized int in the heap
        int *t = new int(7);  // allocates one int in the heap and initializes it with 7
        int *q = new int[7];  // allocates 7 elements (array of size 7) uninitialized in the heap
        q[0] = 7;             // *q = q[0]: once allocated, we can use the pointer as an array
        int x = *q;           // equivalent to: x = q[0]; now both point to the same memory
        delete []q;           // we have to worry about the deleting at the end

        // MEMORY LEAK - example 1
        double *p1 = new double;
        double *p2 = new double[10];
        p1 = p2;   // p1 points now the 10 doubles, the memory of the first double is allocated and non-usable

        // MEMORY LEAK - example 2
        double* function(int a, int b){
            double* p = new double[100];
            double* result = new double[200];
            .. ;
            return result;                    // we should have added: delete[] p;
        }
        double* r = function(1,2);            // after this, we have to add: delete[] r;

        // MEMORY LEAK - example 3
        double *p = new double[27];
        double *p = new double[40];
        delete[] p;  // even if we delete p, the memory of the first 27 doubles is still allocated and unreacheable

//-------------------------------------------
// FUNCTION OVERLOAD
//-------------------------------------------
// Overloaded functions must differ in the number or the type(s) of their parameters.
// * same return type, different parameter lists -> ok
        void print();
        void print(string);
// * different return type, same parameter lists -> ERROR
        int function(const int&);
        bool function(const int&);
// * passing-by-value with "const" -> ERROR: from the point of view of the compiler they are equal
        int function(int);
        int function(const int);
// * passing-by-reference with "const" -> ok
        int function(int&);
        int function(const int&);
// * in-class "const" -> ok
//   (the non-const version will NOT be available for const objects,
//    non-const objects will have both, non-const version will be however a better match)
        class Example{
            public:
                function() const;
                function();
        };
```

```
//---------------------------------------------
// ITERATORS
//---------------------------------------------
// We think of iterators as pointers to access any container.
// Like pointers, iterators give us indirect access to an object.
// Types that have iterators have members that return iteratos: begin() and end().
// The compiler determines the type of b and e
        <container_type> cont;
        auto b = cont.begin();  // b is the pointer to the first element of cont
        auto e = cont.end();    // e is the pointer to the one past the last element of cont
// If the container is empty then the two pointers are equal.

// * every time we access a data structure we have to check if it's empty, now we have an easy way:
        string s("some string");
        if(s.begin() != s.end()){ .. }

// * we can have a easiest way to perform a for-loop:
        string s("some string");
        for(auto it=s.begin(); it!=s.end(); ++it){ .. }

// * we can use iterators as normal pointers
        string s("some string");
        if(s.begin()!=s.end()){
            auto it = s.begin();
            *it = 'c';              // now: s = "come string"
        }

// * operations
        *iter           // we access to the element pointed by iter
        iter -> memb    // we access to an element in a class
        (*iter).memb    // we access to an element in a class
        ++iter          // we move -> by 1
        --iter          // we move <- by 1
        iter1 == iter2  // comparable if: they point to the same object or
        iter1 != iter2  //    they point to the one-past-last-element of the same container type
        iter + n        // we move -> by n NOT changing the iterator
        iter - n        // we move <- by n NOT changing the iterator
        iter += n       // we move -> by n changing the iterator
        iter -= n       // we move <- by n changing the iterator

// * iterators and const iterators:
//      type::iterator it;
//      type::const_iterator it;
        vector<int>::iterator it1;        // read and write iterator
        vector<int>::const_iterator it2;  // only read iterator
        string::iterator it3;             // read and write iterator
        string::const_iterator it4;       // only read iterator
//   however it is convenient to use auto in the case of iterators:
        vector<int> v1;
        const vector<int> v2;
        auto it1 = v1.begin();   // it1 is a vector<int>::iterator
        auto it2 = v2.begin();   // it2 is a vector<int>::const_iterator
        auto it3 = v1.cbegin();  // it3 is a vector<int>::const_iterator

// * we can use iterators to sort
        #include<algorithm>
        vector<type> vec;
        std::sort(vec.begin(), vec.end());

//---------------------------------------------
// SMART POINTERS
//---------------------------------------------
// A pointer initialized with the "new" keyword is pointing to a piece of dynamically allocated memory.
// Hence, we have to delete the pointer explicitely at the end. Freeing dynamic objects can create bugs.
// To avoid problems we use smart pointers, more precisely: shared pointers.

// SHARED POINTERS
        #include<memory>
// Shared pointers are like pointers with an associated counter. Whenever we don't have the pointer pointing to
// an object, the counter becomes zero and the object is deallocated.

// Properties:
// * shared pointers allow multiple pointers to refer to the same object
// * shared pointers are implemented through templates -> we can have a shared pointer to double, int, ..
        shared_ptr<int> p1;       // share_ptr<type> name;
        shared_ptr<string> p2;
// * "make_shared" function: it allocates and initializes an object in dynamic memory, then it returns
//   a shared_ptr that points to that object
        shared_ptr<int> p1    = make_shared<int>(42);        // shared_ptr to an int with value 42
        shared_ptr<string> p2 = make_shared<string>(3,'9');  // shared_ptr to a string with value "999"
        shared_ptr<int> p3    = make_shared<int>();          // shared_ptr to a default init int (=0)
        auto p4 = make_shared<vector<string>>();             // shared_ptr to a empty vector<string>

// * ATTENTION: always check if a pointer is pointing to something (ERROR if we access a non-existing object)
        if(p) { .. }   // equivalent to say "if p is pointing to an object"
// * operations:
        shared_ptr<type> sp      // null smart pointer that can point to objects of type "type"
        sp                       // used as a condition: true if sp points to an object
        *sp                      // access to the object to which sp points
        sp->mem                  // equivalent to (*sp).mem
        sp.get()                 // returns the raw pointer embedded in sp
        swap(p,q)                // (also: p.swap(q)) swaps the pointers p and q,
        make_shared<type> args   // returns a shared_ptr to a dyn. all. object of type "type
```

```cpp
        share_ptr<type> p(q)      // p is a copy of the shared_ptr q
        p = q                     // now also p points to what points q
        p.unique()                // returns true if p is the only one pointing its pointed object
        p.use_count()             // returns the number of pointers that point to *p
// * shared_ptrs automatically destroy their objects and automatically free the associated memory
        shared_ptr<int> function(int n){
                return make_shared<int>(n);
                }

        void use_function(int n){
            shared_ptr<int> p = function(n);
            .. ;
        }   // p goes out of scope and the memory to which p points is automatically freed

        shared_ptr<int> use_function2(int n){
            shared_ptr<int> p = function(n);
            .. ;
            return p;  // the counter gets +1
        }   // p goes out of scope, the counter gets -1, but p still got 1!

// RAW POINTERS vs SHARED POINTERS
// We cannot use a shared pointer to store the address of an existing variable.
// Every time we initialize a smart pointer we create dynamically a new variable.
// We should use raw pointers to store addresses of existing variables, and smart pointers
// if we want to declare a new dynamic variable.

//--------------------------------------------
// Random REMEMBER
//--------------------------------------------
// 1. Passing variables:  * call-by-value           -> small objects
//                        * call-by-const-reference -> large objects
//                        * call-by-reference       -> when we have to return a result
        class Image { .. };
        void function(Image i);         // very slooow
        void function(Image &i);        // here the image will be an INPUT and OUTPUT
        void function(const Image &i);  // here the image will be an INPUT

// 2. Scopes: global constants are okay, global variables are NOT okay.
//    The local redefinition is stronger than the global/outside.

// 3. Pointers of pointers
        int value = 1024;
        int *p = &value;
        int **pp = &p;
        std::cout << "Direct value "          << value;
        std::cout << "Indirect value "        << *p;
        std::cout << "Doubly indirect value " << **pp;

// 4. Auto
//    We can use "auto" only if we're performing an assignment.
//    It is convenient when we have to go through a container, for example
        vector<int> v{1,2, .., 9};
        for(auto& i : v) { .. }      // this may modify the elements of v
        for(auto  i : v) { .. }      // this makes a copy of the elements of v
        for(const auto& i: v){ .. }  // this cannot modify v, however it doesn't copy it either

// 5. When using iterator, we first have to check if they are valid!
//    (which is equivalent to check if a container is empty: always do!)
        auto iter = v.begin();
        if(iter != v.end()){ .. }
```

```
//----------------------------------------------------------------------------------------------------------------------
// 4. INHERITANCE AND POLYMORPHISM
//----------------------------------------------------------------------------------------------------------------------

//---------------------------------------------
// INHERITANCE - pt. 1
//---------------------------------------------
// Inheritance provides a way to create a new class from an existing one.
// The new class is a specialized version of the existing one (code reuse and evolution).
// Classes inherit both data and functions from parent classes.
// "The child (derived) type inherits (is derived from) the parent (base) type".

// A derived class
// * DOES inherit    : data members, member functions
// * DOESN'T inherit : constructors/destructor, assignment operator, friends
// * CAN ADD         : data members, functions (/overwrite them), constructors/destructor
// * SYNTAX          :
        class <derived_class_name> : public <base_class_name> { .. };

// Example
        class Animal{ .. };
        class Dog : public Animal{ .. };

// An object of the derived class
// * has all members of the parent class and of its own class
// * can use all public members of the parent class and of its own

// Members can be:
// * private   : accessible only in the class itself
// * public    : accessible anywhere outside the class
// * protected : accessible in subclasses of the class and inside the class
//               (inherited objects cannot access, only subclasses can access)

// Example
        class Polygon{
            protected:
                int num_vertices;
                float *x_coord, *y_coord;
            public:
                void set(float *x, float *y, int n_ver);
        };

// then writing
        class Rectangle : public Polygon{
            public:
                float area();
        };

// is more convenient than:
        class Rectangle{
            protected:
                int num_vertices;
                float *x_coord, *y_coord;
            public:
                void set(float *x, float *y, int n_ver);
                float area();
        };

// PROTECTED MEMEBERS
// Like private, protected members are inaccessible to objects (of both parents and children).
// However, protected members are accessible to members (and friends) of derived classes.
// We have to make a distinction between objects and classes:
// * protected members from the OBJECT point of view are like PRIVATE
// * protected members from the DERIVED CLASS point of view are like PUBLIC
        class Base{
            protected:
                int protected_member;
        };

        class Derived : public Base{
            private:
                int value;
                void function_1(Derived&);
                void function_2(Base&);
                };

        void Derived::function_1(Derived& d){
                d.value = d.protected_member;    // ok -> a Derived function accessing a private member of a Derived obj
            d.protected_member = 0;          // ok
        }

        void Derived::function_2(Base& b){
            b.protected_member = 0;    // ERROR -> a Derived function accessing a private member of a Base obj,
        }                              // from the Base point of view the Derived functions are external functions

// PRIVATE MEMBERS
// Private members are non-accessible even by the subclasses.
        class Base{
            private:
                int private_member;
            protected:
                int protected_member;
            public:
                int public_member;
        };
```

```
        class Derived : public Base{
            private:
                int function_1(){ return private_member; }   // ERROR -> inherited private members are unaccessible
            public:
                int function_2(){ return protected_member; } // ok
        };

        Derived d;
        int i = d.public_member;     // ok    -> public_member is public in the derived class
        int j = d.protected_member;  // ERROR -> protected_member is unaccessible from derived class' objects
        int k = d.function_1();      // ERROR -> 1st: it's a private function, 2nd: it has its own error
        int m = d.function_2();      // ok

// Notice that, even if the derived class cannot access (directly*) to "private_member", it still has it.
// Inheritance always brings everything from the ancestors: we cannot drop things, we can only add.
// (*directly: because probably there will be getters and setters (that the derived class will inherit))

//------------------------------------------
// POLYMORPHISM
//------------------------------------------
// Polymorphism is the ability of objects to respond differently to a function call.
// An object has "multiple identities" based on its class inheritance tree.
// There are three overwriting methods mechanisms: overloading, redefinition and overriding.
// Overriding provides polymorphism and it's the most powerful mechanism for changing a method behavior.

// REDEFINING BASE CLASS FUNCTIONS
// We can redefine a function in the derived class (mantaining the same name and parameter lists as
// the function in the base class). This is used to replace a function with different actions.
// This mechanism is different from overloading. Objects of the base class use the base class version
// of the function, objects of the derived class use the derived class version of the function.

// OVERRIDING and VIRTUAL MEMBER FUNCTIONS
// The base class can define as "virtual" the functions it expects to be defined directly by its derived classes.
// Without the "virtual" keyword we fall in the case of redefining base class functions.

        class Base{
            private:
                int private_elem = 1;
            protected:
                int protected_elem = 1;
            public:
                Base() = default;
                Base(int v1, int v2): private_elem(v1), protected_elem(v2){}
                int geter_private(){ return private_elem; }
                virtual int function() const{ return private_elem + protected_elem; }
                virtual ~Base() = default;
        };

        class Derived : public Base{
            private:
                int private_elem_2;
            public:
                Derived() = default;
                Derived(int v1, int v2, int v3): Base(v1, v2), private_elem_2(v3){}
                int function() const override;
        };

// * the base class has a default constructor which will initialize private_elem=1 and protected_elem=1
// * the base class has an ad-hock constructor that will initialize private_elem=v1 and protected_elem=v2
// * whenever we use "virtual" -> define a virtual destructor (for security always put a virtual destructor)
// * the derived class constructors rely on the base class constructors
// * the derived class constructor in "Derived.cpp" wuold have been written as:
        Derived::Derived(int v1, int v2, int v3): Base::Base(v1, v2), private_elem_2(v3){}
// * we use the "virtual" keyword only in the body-class declaration, in "Derived.cpp" we have:
        int Derived::function() const{ return private_elem + protected_elem + private_elem_2; }
// * note: a function that is declared "virtual" in the base class is implicitly "virtual" in the derived classes

// DYNAMIC BINDING
// Dynamic binding is the opportunity to use one implementation or another (base/derived) at runtime
// according to the type of the object that we're passing to the function.
// To have dynamic binding we need:     * "virtual"  keyword
//                                      * "override" keyword
//                                      * pointers/references

// Considering the above Base-Derived classes declaration and implementation:
        int final_function(const Base& item){
            return item.function();
        }
// If we pass an object of the base class we use the base class implementation.
// If we pass an object of the derived class we use the derived class implementation.
        Base b(1,2);
        Derived d(1,2,3);
        int i = final_function(b);      // i = 3
        int j = final_function(d);  // j = 6
// The polymorphic behavior is only possible when an object is referenced by a reference/pointer

// FINAL CONSIDERATIONS: DYNAMIC vs STATIC
// Redefined functions are statically bound and overridden functions are dynamically bound.
// A virtual function is overridden, a non-virtual function is redefined.
// To have dynamic binding:  * the member function must be declared virtual in the base class
//                           * the member function must be declared with override in the derived class
//                           * the member function is run through a pointer/reference to a base class object
// A subclass can overwrite (change a class method behavior) in three ways:
// * overloading : same method name, different parameters
// * redefinition: same method name, same parameters (and missing at least one overriding condition)
// * overriding  : same method name, same parameters, all three dynamic binding conditions
```

```
// ABSTRACT BASE CLASSES and PURE VIRTUAL FUNCTIONS
// A virtual member function that MUST be overridden in a derived class MUST have
// NO function definition in the base class. A class become an abstract base class when one
// (or more) of its member is a  pure virtual function.
// An abstract class CANNOT have any objects. It only serves as basis for derived classes.
// To denote a pure virtual function we write:
        virtual void function() = 0;

// Example
// Suppose that we have a bookshop and different policies of discounting.
        class Book{
            protected:
                int book_code;
                int copies;
                double price;
            public:
                Book() = default;
                Book(int code, int num, double pr): book_code(code), copies(num), price(pr) {}
                virtual double total_price() const;
                virtual ~Book() = default;

        };

        class Discount : public Book{
            protected:
                int min_qt;
            public:
                Discount() = default;
                Discount(int code, int num, double pr, int qt): Book(code, num,pr), min_qt(qt){}
                virtual double total_price() const = 0;  // pure virtual
                virtual ~Discount() = default;
        };

        class Disc1 : public Discount{
            public:
                Disc1() = default;
                Disc1(int code, int num, double pr, int qt): Discount(code, num, pr, qt){}
                double total_price() override;
                };

// The class "Discount" only gives a structure for all the possible discounts. There will be no objects.
// In the class "Discount" the function "total_price()" is made pure virtual after being only virtual.
// Since we override the function in "Disc1", we can create objects of type "Disc1".
// Adding "Discount" is an example of refactoring (i.e. redesigning a class hierarchy to move
// operations and/or data from one class to another).

//--------------------------------------------
// INHERITANCE - pt. 2
//--------------------------------------------
// DERIVED-TO-BASE CONVERSION
// Because a derived object contains parts corresponding to its base class(es),
// we can use an object of a derived type as if it were an object of its base type(s).
// We can bind a base-class reference/pointer to the base-class part of a derived object.
        class Base{
            public:
                int value_0;
                int value_1;
        };

        class Derived : public Base{
            public:
                int value_2;
                int value_3;
        };

        Base b;
        Derived d;
        Base* p = &b;     // p points to a "Base" object
        p = &d;           // p points to the "Base"-part of a "Derived" object
        Base &r = d;      // r bounds to the "Base"-part of a "Derived" object
        Derived* q = &b;  // ERROR: can't convert base to derived
        Derived& t = b;   // ERROR: can't convert base to derived
        Base b2(d);       // ok: it uses Base::Base(const Base&) constructor
        b2 = d;           // ok: it uses Base::operator=(const Base&)

// The conversion from the derived to base exists because every derived object contains a
// base-class part to which a pointer or reference of the base-class type can be bound.
// However, there is no automatic conversion from the base class to its derived classes.
// When a part of a derived-object is ignored because the object is treated as a base-object
// (through a reference or a pointer) we say that the object is SLICED DOWN.

// INHERITANCE AND CONTAINERS
// Suppose that we have a derived class ("Derived") which is a specification of a base class ("Base").
// We would like to store elements of both categories in the same container (e.g. book and book+discount).
// We cannot store them in a vector<Derived> because we cannot perform base-to-derived conversion.
// We cannot store them in a vector<Base> because we would lose the derived-new-members.
// -> we use (smart) pointers in containers, not objects!

// If the objects already exist -> RAW POINTERS
        Base b(0,0);
        Derived d(0,0,0,0);
        vector<Base*> basket;
        basket.push_back(&p);
        basket.push_back(&q);

// If the objects are yet to be defined we can use RAW POINTERS, however it's not suggested
// (because we have to remember to delete objects when we finish)
```

```
        vector<Base*> basket;
        basket.push_back(new Base(0,0));
        basket.push_back(new Derived(0,0,0,0));

// If the objects are yet to be defined -> SMART POINTERS
// (here we don't have to remember to delete anything)
        vector<shared_ptr<Base>> basket;
        basket.push_back(make_shared<Base>(0,0));
        basket.push_back(make_shared<Derived>(0,0,0,0));

// After each of the above cases, we can write (e.g.):
        basket[0]->value_0;
        basket[1]->value_3;


// INHERITANCE AND CONSTRUCTORS (/DESTRUCTORS)
// Derived classes MUST have their own constructors (a destructors). When an object of a derived-class
// is created, the base-class constructor is exectured first, followed by the derived-class constructor.
// A derived-class must use the base-class constructor to initialize its base-class parts.
// A base-class should define a virtual destructor. It's good to make a destructor virtual if the class
// could ever become a base-class.


// INHERITANCE AND STATIC MEMBERS
// Static members defined in the base-class are not replicated.
// They can be used by the class, the sub-classes and by every of their objects:
        class Base{
            public:
                static void something();
        };

        class Derived : public Base{
            public:
                void other(const Derived&);
        };

        void Derived::other(const Derived& derived_object){
            Base::something();           // ok: "Base" defines "something()"
            Derived::something();        // ok: "Derived" inherits "something()"
            something();                 // ok: access through this object
            derived_object.something();  // ok: access through Derived object
        };
```

```
//-----------------------------------------------------------------------------------------------------------
// 5. COPY CONTROL
//-----------------------------------------------------------------------------------------------------------
// Container elements are copies: when we use an object to initialize a container or we insert the object
// in it, a copy of that object value is placed in the container (not the object itself).
// Each class defines a new type and defines the operations that objects of that type can perform.
// Classes can control what happens when objects of the class type are copied, assigned, or destroyed.

//--------------------------------------------
// COPY CONSTRUCTOR and ASSIGNMENT OPERATOR
//--------------------------------------------
// COPY-ASSIGNMENT OPERATOR
// It might be convenient to define a copy operator ad-hock:
        class Example{
            private:
                int value;
                std::string text;
            public:
                Example& operator=(const Example&);
                };

        Example& Example::operator=(const Example& rhs){
            value = rhs.value;      // calls the built-in int assignment
            text = rhs.text;        // calls the string::operator=
            return *this;           // return a reference to this object
        }

// Copy initialization uses the copy constructor. Copy initialization happens when we define
// variables using "=", or when we pass an object as an argument to a parameter of non-reference type,
// or when we return an object from a function that has a non-reference return type.

// COPY CONSTRUCTOR
// A constructor is the copy constructor if the first parameter is a reference to the class type.
// The implementation is similar to the standard constructor implementation:
        class Example{
            private:
                int value;
                std::string text;
            public:
                Example() = default;        // default constructor
                Example(const Example&);    // copy constructor
        };

        Example::Example(const Example& rhs): value(rhs.value), text(rhs.text) {}


// RULE OF THUMB
// If a class needs a copy constructor, it almost surely needs a copy-assignment operator (and vice versa).
// If a class needs a destructor, it almost surely need a copy constructor AND a copy-assignment operator.

//--------------------------------------------
// DESTRUCTOR
//--------------------------------------------
// Destructors do whatever work is needed to free the resources used by an object
// and destroy the non-static data members of the object.
// Since we never know if a class will be involved in inheritance, it's always convenient
// to define a virtual destructor of the class:
        virtual ~Example() = default;
        virtual ~Example() {}

//--------------------------------------------
// DEFAULT and DELETE
//--------------------------------------------
// DEFAULT
// We can put everything in default mode:
        class Example{
            public:
                Example() = default;                            // default constructor
                Example(const Example&) = default;              // default copy constructor
                Example& operator=(const Example&);
                ~Example() = default;                           // default destructor
                };
        Example& Example::operator=(const Example&) = default;  // default assignment

// DELETE
// We may want to deny copies or assignemnts (for some reasons).
// Not defining the copy-control members won't prevent the compiler to synthesize them.
// We can prevent copies by defining the copy constructor and copy-ass. operator as deleted functions.
        class Example{
            public:
                Example() = default;                            // default constructor
                Example(const Example&) = delete;               // deleted copy constructor
                Example& operator=(const Example&) = delete;    // deleted assignment
                ~Example() = default;                           // default destructor
        };
// After this, if we try to do "object_1=object_2;" we get a compiler error.
// Note: if we're saying no copies, we're being strict on many things, for example we CANNOT
// have a vector of this class objects (since vectors are containers where objects are copied).
```

```
//------------------------------------------
// COPY CONTROL AND RESOURCE MANAGEMENT
//------------------------------------------
// Classes that manage resources that do not reside in the class must define the copy-control members.
// Classes copy mmbers of buil-in type other than pointers. What a class do when it copies a pointer
// defines whether a class has a like-a-value behavior or a like-a-pointer behavior.

// LIKE-A-VALUE
// Like-a-value classes have their own state. Copies and original are independent.
// In this case is suggested to not use pointers, just to be sure to not create bugs.
// Example: vector of strings
        class Example{
            private:
                std::vector<std::string> data;
        };
// we use a simple vector of strings: when we copy objects of this class we copy the vector
// (two different (copied) objects will be independent one of another).

// LIKE-A-POINTER
// Like-a-pointer classes act like pointers and share part of the state. Copies and original share
// the same underlying data -> changes made to the copy also change the original, and vice versa.
// In this case is suggested to use shared pointers.
// Example: vector of strings
        class Example{
            private:
                shared_ptr<std::vector<std::string>> data;
        };
// we created a shared pointer to a vector of string: when we create objects by copy we do a copy of the
// shared pointer and we obtain two objects, each with its pointer, pointing to the same vector of string.

//------------------------------------------
// IMPLICIT CLASS-TYPE CONVERSIONS
//------------------------------------------
// Constructors that can be called with a single argument define an implicit conversion to a class type.
        class Example{
            private:
                int value_1;
                int value_2;
            public:
                Example() = default;
                Example(int v1): value_1(v1), value_2(0){}
                Example operator+(const Example&) const;
        };

        Example ex=7;      // ok, implicit conversion: ex has value_1=7, value_2=0
        ex=ex+5;           // ok, implicit conversion: ex = ex + [value_1=5, value_2=0]
        void function(Example e){ .. }
        function(7);       // ok, implicit conversion: call function() with value_1=7, value_2=0

// To avoid implicit conversion, whenever we declare a constructor with only one argument we add "explicit":
        class Example{
            private:
                int value_1;
                int value_2;
            public:
                Example() = default;
                explicit Example(int v1): value_1(v1), value_2(0){}
                Example operator+(const Example&) const;
        };

        Example e1=7;      // ERROR: no implicit conversion
        Example e2(7);     // ok: explicit constructors can be used only for direct initialization
        Example e3(5);     // ok
        e3=e3+5;           // ERROR: no implicit conversion
        e3=e3+Example(5); // ok
        void function(Example e){ .. }
        function(7);       // ERROR: no implicit conversion
```

```cpp
//----------------------------------------------------------------------------------------------------------------------
// 6. STL - STANDARD TEMPLATE LIBRARY
//----------------------------------------------------------------------------------------------------------------------

//-------------------------------------------
// SEQUENTIAL CONTAINERS
//-------------------------------------------
// Sequential containers let the programmer control the order in which the elements
// are stored and accessed. The order does not depend on the values of the elements,
// but on their position.

// Sequential containers provide fast sequential access to their elements.
// They offer different performance trade-offs in terms of:
// * costs of adding/deleting elements
// * costs of performing non-sequential access

// * vector       : fast random access, fast insert/delete (only) at the end
// * deque        : fast random access, fast insert/delete (only) at the front/end
// * list         : only bidirectional sequential access, fast insert/delete everywhere
// * forward_list : only sequential access in one direction, fast insert/delete everywhere
// * array        : fast random access, cannot insert/delete
// * string       : only char, fast random access, fast insert/delete (only) at the end

// VECTORS

        #include<vectors>
        using std::vector;

        vector<int> v;
        v.size();         // returns the number of the elements of v
        v.push_back(7);   // add an element with the value 7 to the end of v
        v.resize(10);     // v now has 10 elements
        v = w;            // v is now a copy of w

// * fast random access -> elements are stored contiguously
// * if we add a new element when there's no room left -> the container allocate new memory,
//   copy the elements from the old location into the new space, add the new element and
//   deallocate the old memory
// * to avoid reallocation we can allocate extra capacity at the very first allocation
// * we can characterize a vector with three elements:
//     1. "sz"    = (size) number of elements
//     2. "elem"  = pointer to the first element (in the heap)
//     3. "space" = overall size, how many elements the vector can keep before gettin reallocated

// VECTORS - RESERVE: vector<type>::reserve(unsigned newalloc);
// It focuses only on the memory-block allocation:
// * if the requested size is <= the existing capacity, reserve does nothing
// * after calling reserve, the capacity will be >= the argument passed to reserve
// * if the argument passed to reserve is >= the existing capacity:
//     1. we allocate a block of newalloc elements
//     2. we copy the old elements from the initial data structure into the new block
//     3. we deallocate the old block
//     4. we update the data structure in a way that the pointer of the vector points the new block
// * the complexity is O(sz), because we have to copy element by element

// VECTORS - RESIZE: vector<type>::resize(unsigned newsize);
// It deals with element values:
// * the goal is to reserve newsize elements and fill the "empty" elements in a default manner
// * when the function is called:
//     1. we have a reserve of newsize (we allocate at least newsize elements, possibly more)
//     2. we copy one by one the old elements and we default initialize the remaining elements
//        in such a way that the size of the vector will be newsize
//     3. we deallocate the old block thanks to the functionality in the reserve
// * the complexity is O(newsize), because we have to copy the existing elements (sz)
//   and initialize the remaining ones (newsize-sz)

// VECTORS - PUSH_BAKC: vector<type>::push_back(type value);
// * if there is enough room we simply increment sz and we store the new value
// * if there is NOT enough room we reserve twice sz elements and add the new value
// * the complexity is O(sz), because we have to copy element by element

// VECTORS - FINAL CONSIDERATIONS
// * push_back worst case -> O(sz)
// * push_back average case -> O(1)
// * random access -> O(1)
// * insert != end worst case -> O(sz)
// * insert != end average case -> O(sz)

// WHICH CONTAINER?
// * it is best to use vector unless there is a good reason to prefer another container
// * the program requires random access to elements? -> vector/deque
// * the program requires insert/delete elements in the middle? -> list/forward_list
// * the program requires to insert elements in the middle only while reading inputs and
//   then is needed random access to the elements? -> we can use a list for the input phase
//   and then we can copy the list into a vector
// * the program requires to insert elements in the middle and random access, both constantly?
//   -> evaluate the relative cost of accessing elements in a list/forward_list versus the cost
//   of inserting elements in a vector/deque
// * in general the predominant operation of the application will determine the container
```

```
// CONTAINERS COMMON TYPE AND OPERATIONS
// To make it easy to change the code, we rely on some containers common types:

        container c;           // default constructor, empty container
        container c1(c);       // construct c1 as a copy of c
        container c(b,e);      // copy elements from the range denoted by the iterators b and e
        container c{a,b,..};   // list initalization of the container c
        c.size();              // number of elements of c (NO for forward_list)
        c.max_size();          // maximum number of elements c can hold
        c.empty();             // boolea, True if c has no elements

        // Accessing elements
        // (REMEMBER: always check if the container is empty: if(!c.empty()){ .. }
        c.back();              // returns a reference to the last element of c
        c.front();             // returns a reference to the first element in c
        c[n];                  // NON-CONST reference to the n-th element (only if random access)
        c.at(n);               // CONST reference to the n-th element (only if random access)

        // Example with accessing elements
        if(!c.empty()){
            auto val1 = *c.begin();   // val1 = copy of the value of the 1st element of c
            auto val2 = c.front();    // val2 = copy of the value of the 1st element of c
            auto last = c.end();      // pointer to the one-past-the-last element of c
            auto val3 = *(--last);    // val3 = copy of the value of the last element of c
            auto val4 = c.back();     // val4 = copy of the value of the last element of c
                }

        // Modify the container
        c.inserts(args);        // copy elements as specified by args in c
        c.inserts(p,t);         // creates an elem with value t before the elem pointed by p
        c.insert(p,n,t);        // creates n elements with value t before the elem pointed by p
        c.insert(p,b,e);        // insert the range from b to e before the elem pointed by p
        c.emplace(inits);       // use inits to construct an element in c
        c.emplace(p,args);      // construct from args before the element pointed by p
        c.erase(p);             // remove the element pointed by p
        c.erase(b,e);           // remove elements from the range from b to e
        c.pop_back();           // remove (if exists) the last element of c (NO for forward_list)
        c.pop_front();          // remove (if exists) the first element of c (NO for vector/string)
        c.clear();              // remove all elements from c
        c.push_back(t);         // creates an element with value t and adds it at the end of c
        c.emplace_back(args);   // constructs from args at the end of c
        c.push_front(t);        // same as above, but in the front of c
        c.emplace_front(args);  // same as above, but in the front of c
        c.resize(n);            // resize c to have n elements (if n<c.size() -> deleting)
        c.resize(n,t);          // resize c to have n elements, new elemets (if) have value t

        // Example with insert
        void insert_in_order(list<int>& l, int i){
            if(l.empty()) { l.push_front(i); }
            else{
                auto it = l.begin();
                while(it!=l.end() && *it<i) {
                    ++it;
                                }
            l.insert(it,i);
            }
        }

        // NO for forwar_list
        c.begin();             // returns an iterator to the first element in c
        c.end();               // returns an iterator to the one-past element in c
        c.cbegin();            // as above, but const_iterator
        c.cend();              // as above, but const_iterator
        c.rbegin();            // iterator to the last element
        c.rend();              // iterator to the one past the first element
        c.crbegin();           // as above, but const_iterator
        c.crend();             // as above, but const_iterator

        // Iterators
        *iter           // returns a reference to the element pointed by iter
        iter->memb      // returns a reference to "memb" from the underlying element
        (*iter).memb    // same as above
        ++iter          // increments iter
        --iter          // decrements iter
        iter1==iter2    // comparison: it makes sense only if the iterators are from the same container
        iter1!=iter2    // same as above
        iter+n          // returns the position of iter+n without changing iter
        iter-n          // returns the position of iter-n without changing iter
        iter+=n         // increments iter of n units
        iter-=n         // decrements iter of n units

        // Example for the reverse iterators
        for(vector<int>::const_reverse_iterator it=v.crbegin(); it!=v.crend(); it++) { .. }

        // Assignment operator
        c1 = c2;         // replace the elements of c1 with copies from c2
        c={a,b,..};      // replace the elements of c with copies of elems of the list
        swap(c1,c2);     // exchange elements, here we're NOT copying
        c1.swap(c2);     // same as above
        c.assign(b,e);   // replace elements in c with those in the range denoted by iterators b and e
                         // b and e must NOT be iterators belonging to c
        c.assign(i);     // replace elements in c with those in the initializer list i
        c.assign(n,t);   // replace elements in c with n elements with value t
```

```
// Note: the fact that with "swap" elements are not moved means that iterators, references and pointers
// to the containers are not invalidated. Suppose that we have two vectors of integers, v1 and v2, and
// suppose that we have a pointer, p, to the maximum element of v1. After we perform swap(v1,v2); the
// pointer p still points to the same object, however the object is now the maximum element of v2.
// When do we have POINTER INVALIDATION?
// Suppose we have a pointer, p, which points to the last element of a vector of integers. Suppose that
// we perform a push_back(val); but there's no room left -> a reallocation is performed. The pointer
// p is not considered in the reallocation and so, when the old memory is deleted, p is invalidated.

//-----------------------------------------
// ASSOCIATIVE CONTAINERS
//-----------------------------------------
// Associative containers store their elements based on the value of a key.
// Elements are retrieved efficiently according to their key value.
// Associative containers support the general container operations. However they do not support
// the sequential-container poisition-specific operations (e.g. push_front()).

// ORDERED -> optimized for WORST case complexity
// * map : holds key-value pairs
// * set : the key is the value

// MAP
// Collection of <key,value> paris with unique keys.
// We access to the values through the key:
        map<string, int> word_count;
        string word;
        while(cin >> word){
            ++word_count[word];         // we access to the integer through the key (*)
        }
        for(const auto& w: word_count){
            cout << w.first << "occurs" << w.second << ((w.second>1)? "times":"time");
                }

// (*) if the word is already a key, then the count of that word is incremented,
//     if the word is not yet existing, this will create a pair, assign the key (string)
//     equal to the new word, default initialize the integer (0) and then increment
// * when we want to access the key we use: map_name.first;
// * when we want to access the value we use: map_name.second;
// * the complexity of insert/delete (at worst) is O(log(n)) -> very fast structure
// * the key is CONST
        map<string, int> word_count = .. ;
        auto map_iterator = word_count.begin();
        map_iterator->first    = "something different";   // ERROR
        (*map_iterator).first = "something different";   // ERROR
// * the map types support subscripting
        map_elem[k]     // return the elem with key k, if there's not, it creates it and default init the value
        map_elem.at(k)  // return the elem with key k ONLY IF there's an element with key k
// Example of subscripting:
        map<string, int> word_count;  // empty map
        word_count["Anna"] = 1;       // word_count looks for the element which key is "Anna", the element is
                                      // NOT found, so it adds an element with key "Anna" and default-init value,
                                      // then it changes the value into 1
        word_count.insert(make_pair("Lucia",1));

// PAIR TYPE
// The pair type holds two data members.
// More precisely, pairs are structures with two members, named "first" and "second".
        pair<string, int> p1;
        pair<string, string> p2("Hello", "Hi");
        pair<int, vector<int>> p3;
// * operations:
        pair<type_1, type_2> p1;
        pair<type_1, type_2> p2(v1, v2);
        pair<type_1, type_2< p3 = {v1, v2};
        make_pair(v1, v2);                      // pair definition
        p1.first;                               // returns the 1st member
        p2.second;                              // returns the 2nd member

// SET
// Collection of object. It's useful when we simply want to know whether a value is present.
        set<string> names;
        if(names.find("Mario") == names.end()){ .. }
// * the complexity of insert/delete (at worst) is O(log(n)) -> very fast structure
// * the set structure ignores the replicas, if we construct a set from a vector, the set
//   will keep only the unique values:
        vector<int> vec = {0,0,1,1,2,2};            // vec.size()=6
        set<int> set_vec(vec.begin(), vec.end());  // set_vec.size()=3
// * note: in the above example we could NOT do: set<int> set_vec(vec);
// * since sets are "maps with only key values", every element is CONST:
        set<string> names = .. ;
        auto set_iterator = names.begin();
        *set_iterator = "something different";      // ERROR
// * the set types do NOT support subscripting, since there is no "value" associated with a key
//   (there is no (e.g.): names[k];)
// * include operation: we may want to check if a set B is included in A:
        std::includes(A.cbegin(), A.cend(), B.cbegin(), B.cend());  // bool=1 if True

// MAP and SET
// Example: we don't accept words which are contained in a given set
        map<string, int> word_count;
        set<string> exclude = {"The", "But", "And", "Or", "An", "A",
                               "the", "but", "and", "or", "an", "a"}
```

```
        string word;
        while(cin>>word){
            if(exclude.find(word)==exclude.end())
                ++word_count[word];
        }

// UNORDERED -> optimized for AVERAGE case complxity
// * unordered_map : a map organized by a hash function
// * unordered_set : a set organized by a hash function
// We'll rely only to the already defined hash function.
// This hush function works only for built-in types -> no unordered- for our classes!
// In terms of code, we have (almost) no differences:
        unordered_map<string, int> word_count;
        string word;
        while(cin >> word){
            ++word_count[word];
        }
        for(const auto& w: word_count){
            cout << w.first << "occurs" << w.second << ((w.second>1)? "times":"time");
                }

// ORDERED vs UNORDERED
// The first thing that we look for is: what do we need to optimize?
// If the most frequent operations are find, insert/delete of single elements:
// * MAP           -> optimization of the WORST case complexity (O(log(N)) instead of O(N))
// * UNORDERED_MAP -> optimization of the AVERAGE case complexity (O(1) instead of O(log(N)))

// ASSOCIATIVE CONTAINER TYPE ALIASES
// * key_type    : type of the key of the container
// * mapped_type : type of the map of the container
// * value_type  : same as key_type for set, pair<const key_type, mapped_type> for map
        set<string>::value_type v1;         // string
        set<string>::key_type v2;           // string
        map<string, int>::value_type v3;    // pair<const string, int>
        map<string, int>::key_type v4;      // string
        map<string, int>::mapped_type v5;   // int

// OPERATIONS
        c.insert(v)         // (v value_type object) return pair<iterator,bool>:
                            // iterator to the element with v as key, bool=1 if the elment was inserted
        c.emplace(args)     // same as before, args is used to build an element
        c.insert(b,e)       // b and e are iterators denoting a range of c::value_type elements
        c.insert(l)         // l is a braced list of values
        c.insert(p,v)       // as the first, but uses p as a hint for where to begin the search
                            // for where the new element should be stored
        c.insert(p, args)   // as above
        c.erase(k)          // remove element with key k
        c.erase(p)          // remove element denoted by the iterator p
        c.erase(b,e)        // remove elements in the range denoted by iterators b and e, return e
        c.find(k)           // return an iterator to the first element with value k
        c.count(k)          // return the number of elements which have key k

        // Only for ORDERED
        c.lower_bound(k)    // return an iterator to the 1st elem with key >= k
        c.upper_bound(k)    // return an iterator to the 1st elem with key >  k
```

```
//-------------------------------------------------------------------------------------------------------------------
// 7. MPI - MESSAGE PASSING INTERFACE
//-------------------------------------------------------------------------------------------------------------------

//--------------------------------------------
// PARALLEL PROGRAMMING - THEORETICAL
//--------------------------------------------
// We work in settings of distributed memory: each processor has its own local memory.
// If we want processors to communicate, we exchange data between them.
// There's a limit in the possible speedup thanks to the parallelization (AMDAHL's law):
//     t_n = (f_p/n + f_s) * t_1          t_n = time to run the program on n cpus
//                                        t_1 = time to run the program on 1 cpu
//                                        f_s = fraction of code that is sequential
//                                        f_p = fraction of parallelizable code
// -> the effect of multiple cpus on speed is: speed = t_1/t_n = 1/(f_s + f_p/N)
// However, the real speed-up goes differently due to communication slowdowns:
//     t_n_real = (f_p/n + f_s) * t_1 + k * log_2(n)
// where k represent the communication slowdowns.

// SPMD - Single Program, Multiple Data
// Only a single source code is written, all copies of code start simultaneously and synch periodically.
// DEADLOCK RISK: cpus A and B need both resource R1 and R2. f A gets R1 and B gets R2 we're blocked.

//--------------------------------------------
// MPI - Message Passing Interface
//--------------------------------------------
// Standard structure of every MPI code
        #include<iostream>
        #include<mpi.h>

        int main(int argc, char *argv[]){
            MPI_Init(&argc, &argv);
            int rank, size;
            MPI_Comm_size(MPI_COMM_WORLD, &size);
            MPI_Comm_rank(MPI_COMM_WORLD, &rank);
            .. ;
            MPI_Finalize();
            return 0;
                }

// * if we run the program without arguments -> argc=1, argv[0] is the name of the program
// * if we pass k parameters -> argc=k+1, argv[1] first parameter, .., argv[k] k-th input
// * "MPI_Init(..) provides arguments across all the processes (!arguments are shared with all)
// * after "MPI_Comm_size" -> size = #processors we're using
// * after "MPI_Comm_size" -> rank = ID of the processor
// * "MPI_Finalize()" is saying that we're done with MPI

// COMPILE and RUN
// Compile: either compile the program with a specific executable name or not
        $ mpicxx --std=c++11 file_name.cc -o exe_name                     // 1.
        $ mpicxx --std=c++11 file_name.cc                                 // 2.
        $ mpicxx --std=c++11 file_name.cc file1.cc file2.cc -o exe_name // 3. (e.g. with classes)
        $ mpicxx --std=c++11 file_name.cc file1.cc file2.cc               // 4.

// Note: we add the .cc (.cpp), we don't need to write also the headers

// Run the executable: either with the executable name or no, either adding arguments or not
        $ mpiexec -np=4 exe_name                         // 1.a, 3.a
        $ mpiexec -np=4 exe_name text1 text2             // 1.b, 3.b
        $ mpiexec -np=4 a.out                            // 2.a, 4.a
        $ mpiexec -np=4 a.out text1 text2                // 2.b, 4.b
// * 1.a, 2.a, 3.a, 4.a: running on 4 processors, argc=1, argv[0] is the name of the program
// * 1.b, 2.b, 3.b, 4.b: running on 4 processors, argc=3, argv[1]="text1", argv[2]="text3

//--------------------------------------------
// COMMUNICATORS
//--------------------------------------------
// We can have two types of communications between processors:
// * point-to-point: involves two processors -> functions: sender, receiver
// * collective: involves all processors -> functions: broadcast, reduce, scatter, gather,

// ******************
// SEND and RECEIVE
// ******************

        MPI_Send(const void *buf,                       // where is what we want to send?
                 int count, MPI_Datatype datatype,      // how many (at most) and of what type?
                 int dest, int tag, MPI_Comm comm)      // to who? tag? through what?

        MPI_Recv(void *buf,                              // where do we store?
                 int count, MPI_Datatype datatype,       // how many (at most) and of what type?
                 int source, int tag, MPI_Comm comm,     // from who? tag? through what?
                 MPI_Status *status)                     // "MPI_STATUS_IGNORE"

// Example: suppose that rank0 takes as input a double and wants to send it to the others
        double n;
        if(rank==0){
            cin >> n;
            for(int dest=1; dest<size; ++dest)
                MPI_Send(&n, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
        }
```

```
        else{  // rank!=0
            MPI_Recv(&n, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

// * we declared the variable at the beginning: every rank has to have where to store the info
// * SENDING: where is stored the information? &n. How many of what type? 1 of MPI_DOUBLE.
//          To who are we sending? dest (which varies in the range). Tag? 0. Communicator word.
// * RECIVING: where will be stored the info? &n. How many of what type? 1 of MPI_DOUBLE.
//          From who are we sending? rank_0. Tag? 0. Communicator word and status ignored.
// * rank_0 has to send to all rank the information -> for-loop
// * each rank != 0 has to receive the information just once -> NO for-loop

// MESSAGE MATCHING:     * the communicator word must be the same for receiver and sender
//                       * the destination of the sender must be the source of the receiver
//                       * the tag must be the same for receiver and sender
//                       * the send_type must be the same as the receive_type
//                       * the recv_count must be >= send_count

// *******************
// BROADCAST
// *******************
// We may want rank_0 to send a message to all the other ranks, without doing it one by one.
// We procede with a collective function:
// * cannot use tags
// * every other process in the communicator MUST call the collective function

        MPI_Bcast(void *buffer,                         // sender: where is stored the info
                                                        // receiver: where to store the info
                  int count, MPI_Datatype datatype,     // how many (at most) and of what type?
                  int root, MPI_Comm comm)              // who owns the data? source. Through what?

// Example: suppose that rank0 takes as input a double and wants to send it to the others
        double n;
        if(rank==0)
            cin >> n;
        MPI_Bcast(&n, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
// We don't need for-loops: every rank running this cell either receive (onece) or send (once, to all).

// *******************
// REDUCE/ALL_REDUCE
// *******************
// We may want to perform efficiently some operations among ranks.
// If every rank stores a value and we want the sum of that value, we can send all the values
// to a single rank and then perform the sum there. However this is not efficient (the rank
// performing the sum must wait every other rank to be perfectly aligned).
// It's more convenient if ranks perform partial sums until the very last sum.

        MPI_Reduce(const void *sendbuf, void *recvbuf,  // send buffer, receiver buffer
                   int count, MPI_Datatype datatype,    // how many (at most) and of what type?
                   MPI_Op op, int dest, MPI_Comm comm)  // what operation? send to who? through what?


// Example: suppose that every rank has a local value and we want the sum of them in rank_0
        double local_value;
        double total;
        MPI_Reduce(&local_value, &total, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
// From every rank we're taking local_value and we're adding it into the variable total of rank_0.
// We're performing a sum (-> MPI_SUM), but we may perform: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD.
// Note that we're declaring the variable total in each rank, not only in the destination one.

// What if we want the sum to be saved in the variable total of each rank?

        MPI_Allreduce(const void *sendbuf, void *recvbuf,  // send buffer, receiver buffer
                      int count, MPI_Datatype datatype,    // how many (at most) and of what type?
                      MPI_Op op, MPI_Comm comm)            // what operation? through what?

// MPI_IN_PLACE
// What if every process is storing, for instance, the minimum and we want the overall
// local minimum to be stored in every rank? (without changing the name?)
        double minimum;
        MPI_Allreduce(MPI_IN_PLACE, &minimum, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
// With MPI_IN_PLACE we can use a single buffer for both input and output.
// (we cannot do, for example: MPI_Allreduce(&minimum, &minimum, ...);)

// *******************
// SCATTER and GATHER
// *******************
// When we have to divide data between ranks we can procede in two ways:
// BLOCK PARTITIONING: used when data source is available on a single process
// CYCLIC PARTITIONING: used when data source is available across all processes
// * in the case of block partitioning we have to have: #elements of data multiple of #processes
// * in the case of cyclic partitioning we don't have this problem:
        for(size_t i=rank; i<v.size(); i+=size) { .. }

// Block partitioning is implemented by the function:

        MPI_Scatter(const void *sendbuf,                    // send buffer
                    int sendcount, MPI_Datatype sendtype,   // #elems we send to individual rank? type?
                    void *recvbuf,                          // receiver buffer
                    int recvcount, MPI_Datatype recvtype,   // #elems will be received? type?
                    int root, MPI_Comm comm)                // source of the data? through what?
```

```
// The dual of MPI_Scatter (decomposes one into many) is MPI_Gather (composes many into one):

        MPI_Gather(const void *sendbuf,                    // send buffer
                   int sendcount, MPI_Datatype sendtype,   // #elems we send? type?
                   void *recvbuf,                          // receiver buffer
                   int recvcount, MPI_Datatype recvtype,   // #elems will be received? type?
                   int root, MPI_Comm comm)                // destination of the data? through what?

// If we want the send buffer to be update in all ranks, then:

        MPI_Allgather(const void *sendbuf,                    // send buffer
                      int sendcount, MPI_Datatype sendtype,   // #elems we send? type?
                      void *recvbuf,                          // receiver buffer
                      int recvcount, MPI_Datatype recvtype,   // #elems will be received? type?
                      MPI_Comm comm)                          // through what?

//-------------------------------------------
// READ_VECTOR
// functionality of reading and spreading the informations among the processes
// (after this, the vector "name" will be only local)
// Example: const vector<double> x = mpi::read_vector(n, "x", MPI_COMM_WORD);
//     * after the call, we'll manually insert in the vector x n elements, which will be
//       all initially stored in rank_0's x, and then spread
//     * at the end of the call, each rank will have a local x with n/size elements in it

        vector<double> read_vector(unsigned n, string const& name, MPI_Comm const& comm){
            int rank, size;
            MPI_Comm_rank(comm, &rank);
            MPI_Comm_size(comm, &size);

            const unsigned local_n = n/size;
            vector<double> result(local_n);  // every rank initializes a vector result of size local_n

            if(rank == 0){
                vector<double> input(n);     // we can initialize the vector input only in rank_0
                for(double& e : input)
                    cin >> e;
                MPI_Scatter(input.data(), local_n, MPI_DOUBLE, result.data(), local_n, MPI_DOUBLE, 0, comm);
            }
            else
                MPI_Scatter(nullptr, local_n, MPI_DOUBLE, result.data(), local_n, MPI_DOUBLE, 0, comm);
            return result;
        }

// Since it's a matter of sending (splitting):
// * if we are the sender: both the *sendbuf and *recvbuf matter (rank_0 still has a part of the data)
// * if we are the receiver: only the *recvbuf matters (*sendbuf can be nullptr)

// PRINT_VECTOR
// functionality of building up a single vector (from locals) and print it (rank_0 prints)
// Example: mpi::print_vector(z, n, "Result vector z: ", MPI_COMM_WORD);
//     * all ranks (included rank_0) send their part to rank_0
//     * rank_0 composes the vector and prints it

        void print_vector(vector<double> const& local_v, unsigned n, string const& title, MPI_Comm const& comm){
            int rank, size;
            MPI_Comm_rank(comm, &rank);
            MPI_Comm_size(comm, &size);

            const unsigned local_n = local_v.size();

            if (rank > 0)
                MPI_Gather(local_v.data(), local_n, MPI_DOUBLE, nullptr, local_n, MPI_DOUBLE, 0, comm);
            else{
                vector<double> global(n);
                MPI_Gather(local_v.data(), local_n, MPI_DOUBLE, global.data(), local_n, MPI_DOUBLE, 0, comm);
                cout << title << "\n";
                for(double value : global)
                    cout << value << " ";
                cout << endl;
            }
        }

// Since it's a matter of receiving (composing):
// * if we are the sender: only the *sendbuf matters (*recvbuf can be nullptr)
// * if we are the receiver: both the *sendbuf and *recvbuf matter (rank_0 still needs to send itself its part)

//-------------------------------------------
// Random REMEMBER
//-------------------------------------------
// 1. When inside the code we have that rank_0 takes something in input (e.g. cin>>a>>b>>c;) we do:
        $ mpicxx --std=c++11 file_name.cc
        $ mpiexec -np 4 a.out
        5 6 7
//     In this way we perform the assignment: a=5, b=6, c=7.
//     What if we have 100 cin and we want don't want to write the line "1 2 3 .. 100" every time?
        $ vim input_short                    // this creates a file where we can write "5 6 7"
        $ cat input_short                    // with this we open the file
        5 6 7
        $ mpiexec -np 4 a.out < input_short  // the "<" means "take inputs from .. "
//     What if the output of the file is very long (or, more generally, we want to save it)?
        $ mpiexec -np 4 a.out < input_short > output_short
//     After this we won't get anything as output, but every output will be saved in the file "output_short".
```

```
// 2. There is a difference between the standard input and when the user passes information through cmd line.
//    To get informations through standard input we need to use std::cin and we need rank_0 to take
//    the informations and then share them with other cpus (with broadcast/scatter/send). If instead
//    we get infos through cmd line (user passes them), then the infos are available for all the processes.
//    * STANDARD INPUT: if we have cin (rank_0) in the code -> separate line of inputs in the cmd line
        $ mpicxx --std=c++11 file_name.cc
        $ mpiexec -np 4 a.out
        5 6 7                                // cin>>a>>b>>c
//    * INFORMATIONS FROM USER: if we get something from the arguments -> we write arguments in the cmd line
        $ mpicxx --std=c++11 file_name.cc
        $ mpiexec -np 4 a.out 5 6 7
//      In this situations, in the code ALL processors have these informations, however we need conversions.
//      These informations are stored in argv, which is a vector of strings.
        if(argc == 4){                               // we have arguments -> argc = 1 + #arguments
            unsigned var1 = std::stoul(argv[1]);     // var1 = 5; (unsigned)
            unsigned var2 = std::stoul(argv[2]);     // var2 = 6; (unsigned)
            int      var3 = std::stoi(argv[3]);      // var3 = 7; (integer)
        }

// 3. We can use MPI_Reduce/MPI_Allreduce also on vectors
        int N = 10;
        vector<double> local_x(N), sum(N);
        MPI_Reduce(local_x.data(), sum.data(), N, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```